Devoir à la maison n°1 - Corrigé

Exercice 1 : Le principe du programme repose sur les propriétés arithmétiques suivantes :

(i) Si $a < b$, alors $reste(a,b) = a$
(ii) Si $a \geq b$, alors $reste(a,b) = reste(a-b,b)$

Montrons ces propriétés :

- (i) Si $a < b$, alors on peut écrire $a = 0 \times b + a$, donc le quotient est $q = 0$ et le reste $r = a < b$.

  (ii) Si $a \geq b$, alors notons r le reste dans la division euclidienne de a par b, c'est-à-dire il existe un entier q tel que $\begin{cases} a = bq + r \\ 0 \leqslant r < b \end{cases}$ On a alors $\begin{cases} a - b = b(q-1) + r \\ 0 \leqslant r < b \end{cases}$, où $q-1$ est bien un entier car q est non nul (sinon on aurait $a = r$ donc $a < b$ ce qui est exclu).

  Les conditions $\begin{cases} a - b = b(q-1) + r \\ 0 \leqslant r < b \end{cases}$ prouvent bien que **r** est le reste dans la division euclidienne de $a - b$ par b.

Voici donc le code du programme :

```
def reste(a,b):
    ''' a et b deux entiers naturels, b non nul '''
    if b>a:
        return a
    return reste(a-b,b)
```

Exercice 2 : 1. Algorithme naïf

(a)
```
def Naif(P,x):
    eval=0
    m=len(P)
    for k in range(m):
        eval+=P[k]*x**k
    return eval
```

(b) L'énoncé nous précise bien que n est le degré du polynôme, donc la longueur de la liste où sont stockées les coefficients de P est $m = n+1$.

Dénombrons les opérations élémentaires que nécessite le programme ci-dessus. L'observation importante est que **le coût d'un passage dans la boucle varie à chaque tour de boucle**, car l'exponentiation à la puissance k ne coûte pas la même chose suivant la valeur de k.

Détaillons :

- avant la boucle : 2 affectations
- boucle $k = 0$: 1 affectation, 1 addition, 1 multiplication, exponentiation puissance 0 de coût 1.
- boucle $k = 1$: 1 affectation, 1 addition, 1 multiplication, exponentiation puissance 1 de coût $\alpha\log(1)$
- boucle $k = 2$: 1 affectation, 1 addition, 1 multiplication, exponentiation puissance 2 de coût $\alpha\log(2)$
- ...
- boucle $k = m-1 = n$: 1 affectation, 1 addition, 1 multiplication, exponentiation : $\alpha\log(\mathrm{n})$
- après la boucle : 1 return

Donc $C_1(n) = 2 + (3+1) + (3+\alpha\log(1)) + (3+\alpha\log(2)) + \ldots + (3+\alpha\log(n)) + 1 = 8 + 3n + \alpha\sum_{k=1}^{n}\log(k)$

En utilisant les propriétés de la fonction logarithme, on a donc $\boxed{C_1(n) = 2 + 3n + \alpha\log(n!)}$

2. Algorithme de Horner

(a)
```
def Horner(P,x):
    eval=0
    m=len(P)
    for k in range(m):
        eval=P[m-k-1]+x*eval
    return eval
```

(b) Ici, **le coût de chaque boucle est constant** et égal à 3 (1 affection, 1 addition, 1 multiplication), et la boucle s'exécute $m = n+1$ fois. Le coût de ce programme est donc $C_2(n) = 2+(n+1)\times 3+1 = 6+3n$.

Il s'agit d'une complexité linéaire.

(c) D'après le graphe des croissances comparées, **la complexité linéaire est plus efficace que la complexité $C_1(n)$**.

Au niveau des calculs, l'explication de ce gain en efficacité est la suivante : dans l'algorithme naïf, à chaque tour de boucle on demande à l'ordinateur de calculer x^k, **alors qu'il a déjà calculé x^{k-1} au tour précèdent, mais on n'exploite pas du tout ce résultat** (il « repart de 1 » à chaque fois, sans exploiter le fait que $x^k = x^{k-1}\times x$).

L'algorithme de Horner est justement construit de façon à optimiser les calculs. Il ne comporte pas d'exponentiation, donc est beaucoup plus efficace.

Exercice 3 : Pour bien comprendre cet exercice : il s'agit d'utiliser une méthode numérique de résolution d'une équation différentielle. Ici **la méthode d'approximation est donnée par l'énoncé** : il s'agira d'approcher le calcul d'une intégrale par la méthode des trapèzes.

Il n'y a donc pas lieu d'appliquer la méthode d'Euler dans cet exercice. (on aurait pu faire la résolution numérique avec cette méthode mais ce n'est pas ce que l'énoncé proposait)

1. Soit $k \in [\![0; N-1]\!]$. On utilise l'équation différentielle vérifiée par v_s :

$$v_e - v_s = \frac{1}{\omega_c}\frac{dv_s}{dt}$$

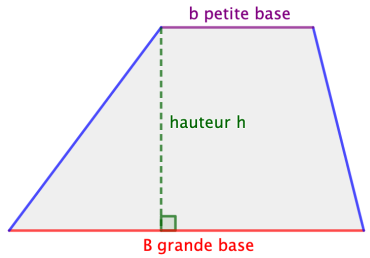
et on intègre cette égalité entre t_k et t_{k+1}.

On a donc $\displaystyle\int_{t_k}^{t_{k+1}} (v_e(t)-v_s(t))dt = \frac{1}{\omega_c}\int_{t_k}^{t_{k+1}} \frac{dv_s}{dt}(t)dt \quad\Leftrightarrow\quad \int_{t_k}^{t_{k+1}} (v_e(t)-v_s(t))dt = \frac{1}{\omega_c}(v_s(t_{k+1})-v_s(t_k))$

Donc : $\boxed{v_s(t_{k+1}) = v_s(t_k) + \omega_c \displaystyle\int_{t_k}^{t_{k+1}} v_e(t) - v_s(t)dt.}$

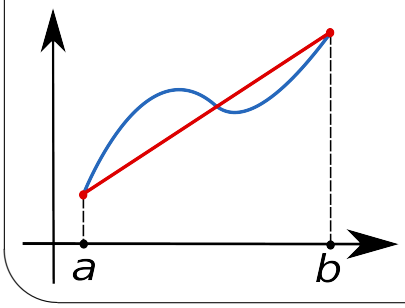
2. Tout d'abord quelques rappels :

Rappel 1 : Aire d'un trapèze



$$Aire = Hauteur \times \frac{(GrandeBase + PetiteBase)}{2}$$

Rappel 2 : la méthode des trapèzes pour le calcul approché d'une intégrale



Principe de la méthode : l'aire sous la courbe représentative de f est approchée par l'aire sous un segment (en rouge).

On a donc : $\displaystyle\int_a^b f(t)dt \approx (b-a) \times \frac{(f(a)+f(b))}{2}$

Soit $k \in [\![0; N-1]\!]$, on approche l'intégrale $\displaystyle\int_{t_k}^{t_{k+1}} (v_e(t)-v_s(t))dt$ par

$$\frac{(t_{k+1}-t_k)(v_e(t_k)-v_s(t_k)+v_e(t_{k+1})-v_s(t_{k+1}))}{2} = \frac{T_e}{2}(v_e(t_k)-v_s(t_k)+v_e(t_{k+1})-v_s(t_{k+1})).$$

On peut alors réécrire

$$\begin{aligned} & v_s(t_{k+1}) = v_s(t_k) + \frac{\omega_c T_e}{2}(v_e(t_k)-v_s(t_k)+v_e(t_{k+1})-v_s(t_{k+1})) \\ \Leftrightarrow & \left(1+\frac{\omega_c T_e}{2}\right) v_s(t_{k+1}) = \left(1-\frac{\omega_c T_e}{2}\right) v_s(t_k) + \frac{\omega_c T_e}{2}(v_e(t_k)+v_e(t_{k+1})) \\ \Leftrightarrow & v_s(t_{k+1}) = \frac{1-\frac{\omega_c T_e}{2}}{1+\frac{\omega_c T_e}{2}} v_s(t_k) + \frac{\frac{\omega_c T_e}{2}}{1+\frac{\omega_c T_e}{2}}(v_e(t_k)+v_e(t_{k+1})) \end{aligned}$$

qui est bien de la forme $\boxed{v_s(t_{k+1}) = Av_s(t_k) + B(v_e(t_{k+1})+v_e(t_k))}$ avec

$$A = \frac{1-\frac{\omega_c T_e}{2}}{1+\frac{\omega_c T_e}{2}} \quad \text{et} \quad B = \frac{\frac{\omega_c T_e}{2}}{1+\frac{\omega_c T_e}{2}}.$$

3. On suppose que les variables A et B sont déclarées comme des ***variables globales***

```
1 def signsortie (Ve):
2     Vs=[1] # le tableau des signaux de sortie
3     for k in range(len(Ve)-1): # on a déjà renseigné le premier élément
4         Vs.append(A*Vs[k]+B*(Ve[k+1]+Ve[k]))
5     return Vs
```